TECHNISCHE UNIVERSITÄT BRAUNSCHWEIG

Abteilung Entwurf integrierter Schaltungen (E.I.S.)
Mühlenpfordtstraße 23
38106 Braunschweig

Seminar

Embedded Systems - Entwurfsmethoden und Anwendungen

Embedded Java

Frank Eichinger, B.Sc.
Matr.-Nr. 2609977

Professor	:	Prof. Dr. Ulrich Golze
Betreuer	:	Dipl. Inform. Wolfgang Klingauf
Eingereicht am	:	19. Januar 2004

Kurzfassung

In dieser Seminararbeit geht es darum, die Vorzüge und Nachteile von der Programmiersprache Java bei der Entwicklung von eingebetteten Systemen aufzuzeigen. Es werden zunächst die grundlegenden Probleme des Softwareentwurfs für eingebettete Systeme diskutiert, bevor auf mögliche Java-Lösungen dieser - und auf die dabei entstehenden Herausforderungen - eingegangen wird. Der Schwerpunkt liegt auf der Vorstellung der Java 2 Micro Edition (J2ME) und von JControl, einer Java-Lösung für kleine eingebettete Systeme speziell in der Home-Automation. Des weiteren wird in einer Technologieübersicht auf andere Lösungsansätze, wie verschiedenartige Kompiler und Java-Prozessoren, eingegangen und ein kleiner Ausblick auf zukünftige Entwicklungen gewagt.

Abstract

This seminar paper presents the advantages and disadvantages using the Java programming language for embedded systems. At first, the basic problems of software development for embedded systems are discussed. After that, the advantages of Java as possible solutions for these problems are presented, including the resulting challenges. The main part is the presentation of two products, the Java 2 Micro Edition and JControl, a Java implementation for small embedded systems, especially in the area of home automation. Finally, other Java solutions and products, like different compiler technologies and Java processors, are summarized in a technology overview. The seminar paper ends with a short prediction about the usage of Embedded Java in the future.

Inhaltsverzeichnis

1 Einleitung

Heutzutage sind etwa 98% der Prozessoren nicht in klassischen PC, Workstations oder Servern zu finden, sondern in eingebetteten Systemen aller Art (vgl. [1]). Wie in diesem Kapitel gezeigt werden soll, gestaltet sich der Softwareentwurf für eingebettete Systeme allerdings im Vergleich zur klassischen Softwareentwicklung, z.B. für Desktop Systeme, als unerwartet schwierig. Das mag daran liegen, dass bisher das Hauptaugenmerk bei der Entwicklung eingebetteter Systeme eher bei technischen als bei softwaretechnischen Aspekten lag.

Die Verwendung der jungen Programmiersprache Java scheint nach Überwindung verschiedener Probleme viele Vorteile zu bringen. Da es inzwischen auch leistungsfähige Java-Implementierungen für kleine Prozessoren gibt, kann man auch in diesem Bereich von Java profitieren. Die Vorteile und Probleme von Java für eingebettete Systeme, aber auch konkrete Lösungen sollen in dieser Ausarbeitung diskutiert und vorgestellt werden. Es soll aber schon an dieser Stelle gesagt sein, dass Java-Technologie, wenn sie bei eingebetteten Systemen verwendet wird, andere Technologien im Regelfall nicht ablösen, sondern ergänzen soll.

1.1 Probleme beim Entwurf eingebetteter Software

Der Markt für eingebettete Systeme ist in der letzten Zeit stark am Wachsen. Wie überall ist es auch hier so, dass immer schnellere und leistungsfähigere Geräte zu einem möglichst geringen Preis gefordert werden. Diese Bedingung hat dazu geführt, dass immer mehr Spezialprozessoren verwendet werden, die die für eine spezielle Anwendungsdomäne bestimmten Aufgaben erfüllen können. Diese Funktionalität, z.B. spezielle Schnittstellen oder Kommunikationshardware, wird bei konventionellen Architekturen außerhalb des Prozessors realisiert. Es werden also immer weniger kleine universelle Prozessoren verwendet und es gibt eine große Palette spezialisierter Prozessoren, so dass man für ein eingebettetes System relativ leicht einen optimalen Prozessor finden kann. Die besonders hoch integrierten Prozessoren, die nahezu keine anderen Hardwarekomponenten benötigen, werden auch systems-on-chip genannt. Die Entwicklung hin zu speziellen Prozessoren erfordert allerdings auch die Entwicklung sehr spezieller Software. Es ist ein Problem, dass die Softwaretechnik in den letzten Jahren nicht ganz der kontinuierlichen Komplexitätssteigerung im Hardwarebereich nachkommt. Man spricht hier von der daraus resultierenden productivity gap, welche mit herkömmlicher Assembler- und C-Technologie nur schwer geschlossen werden kann.

Bei eingebetteten Systemen, die nur sehr kleine Prozessoren bzw. Controller einsetzen, kann man heutzutage im hardwarenahen Bereich nicht immer auf fertige Softwarebibliotheken zurückgreifen, welche dann selbst geschrieben werden müssen. Das liegt vor allem an der komplexen und sehr unterschiedlichen Hardware. Genauso verhält es sich natürlich bei der Wiederverwendung von Softwarekomponenten aus bereits bestehenden Entwicklungen, wenn bei einem neuen Projekt andere Hardware eingesetzt wird. Das Problem von nicht passenden Bibliotheken und von ggf. nicht portablen Softwarekomponenten ist u.a. auf den Mangel an Standards für Schnittstellen und Softwarekomponenten kleiner eingebetteter Systeme zurückzuführen. Am häufigsten wird hier in Assemblersprachen oder C entwickelt, wobei ANSI C meist der kleinste gemeinsame Nenner ist.

Der Mangel an Standards und die hardwarenahe Programmierung stellen besonders auch für die Wartung von Software, sowie für die Änderung oder Erweiterung nach Abschluss der Planungsphase des Produkts, große Probleme dar.

2 Java und eingebettete Systeme

In diesem Kapitel soll auf die Vorzüge von Java als mögliche Lösung einiger der im letzten Kapitel genannten Probleme eingegangen sowie die daraus resultierenden Probleme und Herausforderungen besprochen werden.

2.1 Die Vorzüge von Java

Es ist nicht nur die Objektorientierung und die einfachere Programmierung, die Java interessant für eingebettete Systeme macht. Es ist auch die bisher vermisste Möglichkeit, Software unabhängig von der verwendeten Hardware, also ohne Spezialkenntnisse des verwendeten Prozessors, anderer Spezialhardware oder Systemfunktionen des Betriebssystems, generisch zu entwickeln. Java bietet dem Entwickler eine Zwischenschicht zur Hardwareabstraktion an, also eine Art Middleware, zwischen der mitunter komplizierten Systemansteuerung und der gewohnten objektorientierten Softwareentwicklung. Diese Zwischenschicht wird durch strukturierte, einheitliche Schnittstellendefinitionen repräsentiert. Dazu kommt das Java-Konzept „write once run everywhere“, also die Plattformunabhängigkeit. Ein Java-Programm sollte auf jeder Plattform laufen, auf der eine entsprechende virtuelle Maschine verfügbar ist, die Java-Bytecode interpretiert. So wird es z.B. möglich, einen Prozessor durch einen leistungsfähigereren auszutauschen, ohne die Software für ein Produkt neu schreiben zu müssen.

Java bietet von Haus aus eine Reihe von eingebauten Features, die bei anderen Programmiersprachen nicht unbedingt zum Standard gehören. Dazu gehören vor allem die automatische Speicherverwaltung (Garbage Collector) und das Multithreading, welches es erlaubt, mehrere leichtgewichtige Prozesse (Threads) gleichzeitig auszuführen.

Für die Programmierer selbst gibt es noch eine Reihe von weiteren Vorteilen. Zu diesen gehören an erster Stelle die Objektorientierung, aber auch die Verfügbarkeit von vielen Tools und Entwicklungsumgebungen, die zudem oft kostenfrei erhältlich sind. Außerdem angenehm für den Entwickler ist die Einfachheit und Effizienz der Sprache, welche eine schnellere und fehlerfreiere Programmierung als z.B. C ermöglicht. Verschiedene Studien (z.B. [2]) haben gezeigt, dass aus diesen Gründen die Programmierung mit Java sehr produktiv und damit kostengünstig ist, zudem existiert ein inzwischen recht großer Entwicklerpool. So dürfte es deutlich leichter fallen, einen Java-Programmierer oder auch eine Fremdfirma zu finden (vgl. [3]), die Java-Applikationen für ein eingebettetes System entwickelt, als einen Experten für die Programmierung

solcher Systeme in anderen Sprachen, der ein großes technisches Spezialwissen haben muss.

Für Java stehen außerdem umfangreiche Klassenbibliotheken zur Verfügung und durch Objektorientierung und klare Schnittstellendefinitionen fallen Konzepte wie Design- und Code Reuse deutlich leichter. Dadurch, und durch die Verfügbarkeit von entsprechenden Tools, ist außerdem die Vorgehensweise des Rapid Prototyping unter Java viel eher in Betracht zu ziehen, als bei der Verwendung von herkömmlichen hardwarenahen Programmiersprachen.

2.2 Besondere Herausforderungen und Probleme

Wenn man Java als Programmiersprache bei eingebetteten Systemen verwenden möchte, treten verschiedene Probleme auf, die man bei Desktop Rechnern nicht unbedingt findet.

2.2.1 Performance-Aspekte und Ressourcen

Am häufigsten sind die Vorbehalte bzgl. der Performance von Java zu nennen. In der Tat fordern Java-Programme z.B. bei der J2SE eine recht hohe Rechenleistung und sind sehr speicherhungrig. Im Desktop- und vor allem im Serverbereich hat man diese Probleme inzwischen in erster Linie durch immer leistungsfähigere Hardware, aber auch durch effizientere Java-Implementierungen, weitgehend in den Griff bekommen. Im Bereich eingebetteter Systeme würde eine Java-Implementierung, die den vollen Funktionsumfang zur Verfügung stellt, allerdings kaum zu realisieren sein. Deshalb muss hier das Performanceproblem neben effizienten Implementierungen der virtuellen Maschinen durch Einschränkung des Sprachumfangs und der Klassenbibliotheken gelöst werden.

Weitere Ansätze zur Performancesteigerung von Java-Implementierungen befassen sich mit der Erstellung von Java-Prozessoren, die direkt Bytecode ausführen können oder mit der Kompilierung von Java-Code bzw. Klassen oder Methoden zu nativem Code, was den bei Java üblichen und aufwändigen Schritt der Interpretierung des Bytecodes erspart.

Bei der Ausführungsgeschwindigkeit sind C- oder Assembler-Lösungen in den meisten Fällen trotzdem schneller als Java. Da aber auch Prozessoren für eingebettete Systeme inzwischen recht kostengünstig zur Verfügung stehen, kann eine Entscheidung für eine Entwicklungsumgebung aber durchaus auch nach softwaretechnischen Gesichtspunkten getroffen werden und wirtschaftliche Gründe müssen nicht mehr oberste Maxime sein (vgl. [1]).

2.2.2 Speichermanagement und Echtzeit

Der bereits erwähnte Garbage Collector, der in unregelmäßigen Abständen den Speicher von nicht länger benötigten Objekten befreit, führt zu einem nichtdeterministischen Verhalten der interpretierten Java-Programme. Dieses kann durch die nebenläufige Ausführung von Prozessen (mehrere Threads) sowie durch die bei Java übliche Verfahrensweise des Nachladens von Programmteilen bzw. Objekten noch verstärkt werden. In der Praxis bedeutet das, dass zu unvorhersehbaren Momenten Verzögerungen in der Programmausführung entstehen können. Bei vielen Anwendungen stellt das kein großes Problem dar. Beispielsweise sind kurze Verzögerungen bei Benutzerschnittstellen meist vertretbar und auch bei einigen anderen Anwendungen, z.B. in der Home-Automation, kommt es nicht immer auf die exakte Einhaltung von zeitlichen Schranken an. Bei kritischen Echtzeitanwendungen wie z.B. Steuerungen von Flugzeugen, Airbags oder medizinischen Geräten ist ein solcher Nichtdeterminismus aber nicht hinnehmbar.

Bei der Echtzeit wird zwischen harter und weicher Echtzeit unterschieden (hard und soft realtime). Beide Varianten liefern korrekte Ergebnisse, allerdings mit unterschiedlichen zeitlichen Vorgaben. Harte Echtzeitsysteme werden bei zeitkritischen- und sicherheitsrelevanten Aufgaben eingesetzt, hier müssen definierte zeitliche Schranken (deadlines) unter allen Umständen eingehalten werden. Bei weicher Echtzeit ist die Einhaltung solcher Schranken nicht zwingend gefordert. Es führt also zu Verzögerungen oder es sind Lösungen denkbar, bei denen z.B. die Qualität einer Berechnung wie einer Videokompression kurzzeitig herunter gesetzt wird, um eine Zeitvorgabe einzuhalten, allerdings bei verminderter Qualität des Ergebnisses (vgl. [1]). Solche Modifikationen sind bei harter Echtzeit allerdings auf keinen Fall erlaubt.

Durch die bereits erwähnte Speicherverwaltung und dem daraus resultierenden Nichtdeterminismus können mit Java keine harten Echtzeitsysteme realisiert werden. Harte Echtzeit wird also wahrscheinlich auch in der Zukunft eine Domäne von Assembler und C bleiben, aber auch Hybridlösungen sind denkbar, bei denen z.B. konventionelle Programmiersprachen die Steuerung übernehmen und Java für die Benutzerschnittstelle eingesetzt wird. Es gibt allerdings auch verschiedene Ansätze, die Echtzeitprobleme von Java zu lösen und so doch in einem gewissen Rahmen Echtzeitanwendungen mit Java zu realisieren. Zu nennen ist hier in erster Linie die vom Java Community Process (JCP, [4]) entwickelte Real-Time Specification for Java (RTSJ, [5], [6]), die durch Modifikationen und Erweiterungen in sieben unterschiedlichen Bereichen in den Definitionen der Sprache und der virtuellen Maschine, Java echtzeittauglich machen will (vgl. [7]). Beispielsweise werden hier neue Speicher-

bereiche außerhalb des Java-Heap-Speichers eingeführt, die nicht vom Nichtdeterminismus der Java-Speicherverwaltung betroffen sind.

2.2.3 Verschiedenartige Geräte

Ein weiteres Problem entsteht durch die Unterschiedlichkeit von verschiedenartigen Systemen, welche das Java-Prinzip der Plattformunabhängigkeit einschränkt. Bei Desktop-Systemen mit Universalprozessor kann ein Programmierer eine bestimmte Ausstattung voraussetzen, die die Eingabe- und Ausgabegeräte betreffen, beispielsweise Maus, Tastatur und normalgroßer Monitor. Bei Applikationen für eingebettete Systeme kann man Voraussetzungen an die Ausstattung nur dann stellen, wenn man gleichzeitig in Kauf nimmt, dass die Anwendungen nicht auf allen Geräten laufen - das ist allerdings unumgänglich bei eingebetteten Systemen. Allerdings werden in der J2ME verschiedene Profile und Konfigurationen festgelegt, die ein Mindestmaß an Ausstattung festlegen. So kann man Programme für ein bestimmtes Profil schreiben, welche dann auf allen Geräten laufen, die dieses Profil unterstützen. Allerdings kann die Darstellung und z.B. die Anordnung von Tasten dann doch von Gerät zu Gerät unterschiedlich sein. Da so in vielen Fällen Kompromisse eingegangen werden müssen, zeigt die Praxis, dass gerätespezifische Anwendungen meist komfortabler zu benutzen sind, aber eben auf Kosten der Plattformunabhängigkeit.

3 Konkrete Produkte

In diesem Kapitel sollen zwei konkrete Produkte vorgestellt werden. Zunächst die verbreitetste Produktpalette, die Java 2 Micro Edition von Sun, dann JControl, eins von mehreren kleineren Produkten, welches aus einem universitären Projekt hervorgegangen ist.

3.1 Die Java 2 Micro Edition (J2ME)

Sun Microsystems, die Erfinder von Java, decken mit ihrer Java 2 Plattform nahezu die gesamte Breite von Anwendungsmöglichkeiten ab, die man sich

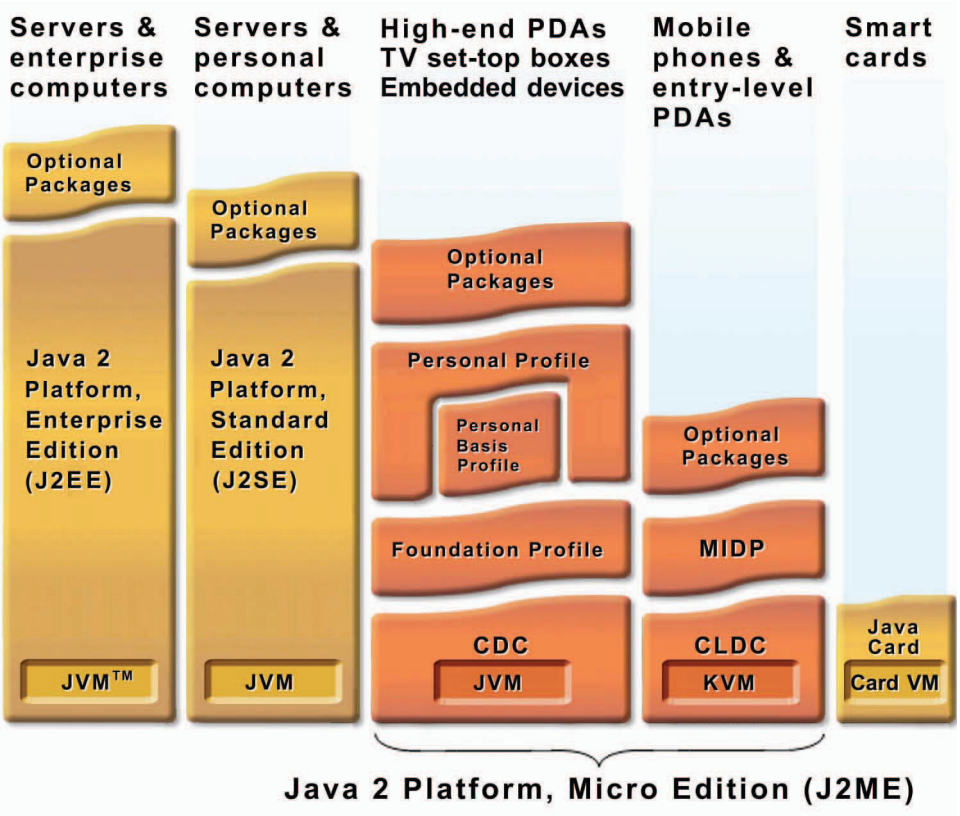

Abbildung 1: Die SUN Java 2 Plattform (entnommen aus [8])

vorstellen kann. Wie man in der Grafik sehen kann, unterteilt sie sich in die folgenden Bereiche:

- die *Java 2 Enterprise Edition (J2EE, [9])*, mit der z.B. verteilte unternehmensweite Anwendungen mit Einbindung von ERP-Systemen möglich sind,

- die *Java 2 Standard Edition (J2SE, [10])*, mit der alle Formen von Desktop-Applikationen sowie in Webseiten eingebettete Applets entwickelt werden können, die
- die *Java 2 Micro Edition (J2ME, [11])*, die in diesem Kapitel genauer vorgestellt wird und zur Entwicklung von eingebetteten Systemen vom Mobiltelefon bis zum großen PDA gedacht ist, und letztendlich
- *Java Card* ([12]), was als Lösung für in Chipkarten integrierte eingebettete Systeme vorgesehen ist.

Wie man in der Grafik erkennen kann, ist die Micro Edition in bisher 2 Konfigurationen unterteilt (CDC und CLDC). Diese gliedern sich wiederum in ein oder mehrere Profile. Das Konzept der Konfigurationen teilt Geräte auf Grund von Hardwareeigenschaften in verschiedene Gruppen ein und legt Eigenschaften wie den unterstützten Sprachumfang, die virtuelle Maschine sowie die minimale Unterstützung von Klassenbibliotheken fest. Profile bauen auf den Konfigurationen auf und spezifizieren die zur Verfügung stehenden Klassenbibliotheken noch genauer indem sie diese erweitern. Ein Programm, das für ein bestimmtes Profil geschrieben wurde, also auch einer bestimmten Konfiguration zugeordnet ist, sollte auf allen Geräten laufen, welche das jeweilige Profil unterstützen, auch wenn sich Darstellung und Eingabemöglichkeiten leicht unterscheiden können. Konfigurationen und Profile werden vom Java Community Prozess (JCP, [4]) spezifiziert und sind somit ein offener Standard an dem alle beteiligten Firmen mitwirken. Die Optional Packages, die Erweiterungen für bestimmte Funktionalitäten für bestimmte Konfigurationen zur Verfügung stellen, werden ebenfalls vom JCP spezifiziert.

3.1.1 Die Connected Device Configuration (CDC)

Geräte, die zur CDC ([13]) gehören, sind laut Sun u.a. TV Set-Top-Boxen, Router, Fahrzeuginformationssysteme und vor allem High-End PDAs (vgl. [14]). Die Geräte sollten über einen 32-Bit Prozessor verfügen und eine Speicherausstattung von 2 MB aufweisen[1]. Es handelt sich also um die eher größeren eingebetteten Systeme und nicht um kleine Steuerungen oder Mobiltelefone. So bietet die hoch optimierte virtuelle Maschine, die CDC HotSpot Implementation[2], auch fast alle Features einer virtuellen Maschine der J2SE und setzt dabei die gleiche Spezifikation um. Beispielsweise unterstützt die

[1] [15] spricht von mindestens insgesamt 768 kB, [8] und [14] von 2 MB und [16] von 2 MB RAM plus 2 MB ROM

[2] ehemals CVM, in der Grafik nur JVM genannt

CDC HotSpot Implementation auch Gleitkommaberechnungen, was bei kleineren virtuellen Maschinen meist nicht der Fall ist.

Zur CDC gehören bislang 3 Profile, welche für unterschiedliche Anwendungsszenarien gedacht sind. Das Foundation Profile ist das kleinste Profil der CDC, es erweitert diese und unterstützt IO und Netzwerkfunktionen, jedoch keinerlei Grafik- oder GUI-Anwendungen. Es ist als Grundlage für weitere Profile gedacht. Das Personal Basis Profile erweitert das Foundation Profile, u.a. um Teile des Abstract Windowing Toolkits (AWT) und um Java Beans und bildet die Basis für das am meisten genutzte Profil der CDC, dem Personal Profile. Dieses unterstützt AWT-GUIs vollständig und bietet Applet-Unterstützung. Ferner kann es auch zur Migration von früheren PersonalJava Applikationen benutzt werden.

3.1.2 Die Connected Limited Device Configuration (CLDC)

Die CLDC ([17]) ist die kleinere Variante der CDC, zu ihr gehören in erster Linie Mobiltelefone und kleinere PDAs, aber auch weitere Geräte wie Waschmaschinen oder solche aus dem Bereich der Home-Automation lassen sich der CLDC zuordnen (vgl. [18]). Technisch betrachtet gibt es bei der CLDC keine konkreten Vorgaben, bis auf Mindestanforderungen an den Speicher. Die offizielle Spezifikation ([19]) sieht hier mindestens 160 kB nichtflüchtigen Speicher (z.B. ROM oder Flash) für die virtuelle Maschine und die Klassenbibliotheken, sowie mindestens 32 kB RAM zur Laufzeit vor, in dem dann z.B. der Objekt Heap Platz findet. Als Prozessoren kommen bei der CLDC typischerweise 16- oder 32-Bit RISC- oder CISC-Modelle zum Einsatz, typischerweise mit einer Taktfrequenz von 12 bis 35 MHz. Der Leistungsbedarf der Geräte soll für den Batteriebetrieb optimiert sein und die oft drahtlose Netzwerkverbindung ist nur sehr schmalbandig, wird aber nicht vorausgesetzt.

Als virtuelle Maschine wird bei der CLDC die KVM verwendet[3]. Das K steht hierbei für Kilo, was ein Zeichen dafür sein soll, dass diese virtuelle Maschine nur mit Speicher im Kilobyte-Bereich arbeitet und daher sehr schlank ist. Im Gegensatz zur JVM der J2SE und der CDC HotSpot Implementation ist die KVM tatsächlich sehr eingeschränkt - es existiert beispielsweise keine Unterstützung für das Java Native Interface, es gibt Einschränkungen bei der Fehlerbehandlung und Gleitkommaarithmetik (Datentypen `float` und `double`) wird nicht unterstützt. Falls man auf Gleitkommaberechnungen nicht verzichten kann, können solche über Bibliotheken anderer Anbieter realisiert

[3] diese wurde zunächst unter dem Namen Spotless entwickelt und soll durch die CLDC HotSpot Implementation abgelöst werden, die dann allerdings nur noch auf 32-Bit Prozessoren läuft; vgl. [20]

werden. Solche Bibliotheken bilden dann die entsprechenden Datentypen auf 32-Bit Integerzahlen ab, was natürlich für die Performance nicht so gut ist.

Die KVM ist ein klassischer Bytecode-Interpreter, der den Java-Bytecode interpretiert und nicht compiliert, wie es im Desktop Bereich längst üblich ist. Eine Besonderheit von CLDC-Anwendungen ist ein anderer Weg der Verifizierung der Klassendateien. Diese Verifizierung findet hier nicht wie bei der J2SE zur Laufzeit statt, da diese Prozedur einen zu hohen Speicherbedarf hat, sondern vorab auf einem Desktop System oder Server, bevor die Applikation in Form eines JAR-Archivs auf das eingebettete System übertragen wird. Diese Vorab-Verifizierung übernimmt ein spezielles Tool, welches ein besonderes Attribut an die Klassendatei anhängt. Mit Hilfe dieses Attributs kann eine schnelle und speichersparende Verifizierung auf dem Gerät vor jeder Ausführung stattfinden.

An Profilen steht der CLDC im wesentlichen das inzwischen bei Mobiltelefonen sehr verbreitete Mobile Information Device Profile (MIDP, [21]) zur Verfügung, welches aber auch für andere Geräte wie kleine PDAs eingesetzt werden kann. Das MIDP bietet ein Benutzerinterface, welches speziell für MIDP-Applikationen, welche Midlets genannt werden, entwickelt wurde. Es stellt im GUI-Bereich Basisfunktionen wie Textfelder, Fenster und Scrollbalken zur Verfügung, welche aber nicht exakt platziert werden. Die Elemente werden also nur angegeben und bei der Ausführung wird dann eine gerätespezifische Anordnung vorgenommen, bei der die Besonderheiten eines jeden Geräts, also z.B. Displaygröße und Eingabeelemente wie Joysticks, berücksichtigt werden können. Des weiteren gehören zur MIDP Klassenbibliotheken zur Grafik- und Musikausgabe, sowie vor allem zur Kommunikation, u.a. über HTTP, HTTPS, Sockets und SMS.

Zur Zeit werden die allermeisten Midlets für kleine Spiele auf Mobiltelefonen entwickelt. Neben Denk- und Geschicklichkeitsspielen sind es vor allem jump-and-run und ähnliche Spiele wie z.B. das bekannte Super Mario World, welche sich zur Zeit einer großen Beliebtheit erfreuen. Java hat somit mit der J2ME und speziell der CLDC und dem MIDP einen einheitlichen Standard für mobile Applikationen auf Mobiltelefonen geschaffen. Durch diesen Standard ist ein großer Markt entstanden und sehr viele Firmen wollen an diesem verdienen. So gibt es viele Anbieter, die MIDP-Spiele kostenpflichtig zum Download anbieten, was oft über den SMS initiiert und abgerechnet wird.

3.1.3 Optional Packages

Optional Packages erweitern die Funktionalität von bestimmten Konfigurationen in Form von weiteren Klassenbibliotheken. Sie sind also kein fester Bestandteil einer Konfiguration oder eines Profils, sondern können bei Bedarf benutzt werden. U.a. stehen Optional Packages für folgende Anwendungsbereiche zur Verfügung:

Bluetooth Kommunikation	
Java Database Connectivity (JDBC)	Erlaubt es, auf (entfernte) relationale Datenbanken zuzugreifen.
Java Remote Message Invocation (RMI)	Ermöglicht es, Remote Procedure Calls (RPCs), also Methodenaufrufe über ein Netzwerk, auf anderen Maschinen durchzuführen.
Web Services[4]	Erlauben es, z.B. über HTTP auf XML-, SOAP- und WSDL basierte Web Services zuzugreifen, also auf eleganten, zeitgemäßen und plattformunabhängigen Weg ebenfalls Methoden oder Funktionen entfernter Hosts aufzurufen und so z.B. 3/4-Tier Anwendungen umzusetzen. Durch dieses Optional Package, welches auch die Integration mit Microsofts .NET Plattform ermöglicht, werden auch minimale Funktionen zur XML-Bearbeitung in der J2ME zur Verfügung gestellt.

3.1.4 Praxisbeispiel: Midletentwicklung

Um ein Midlet, also eine Applikation für das MIDP der CLDC, zu entwickeln, muss als Voraussetzung ein Java Development Kit (JDK) der J2SE installiert sein, ferner benötigt man die Klassenbibliotheken der Konfiguration und des Profils, die von den jeweiligen Homepages ([17], [21]) heruntergeladen werden können. Anstelle der beiden Downloads bietet sich auch das J2ME Wireless Toolkit (WTK, [22]) an, welches neben den Klassenbibliotheken auch noch verschiedene Tools wie die KToolbar sowie Emulatoren enthält.

Um ein Midlet zu programmieren, muss zunächst die Klasse `MIDlet` aus `javax.microedition.midlet` erweitert werden. Ähnlich wie Applets oder Servlets haben Midlets keine `main()` Methode, sondern eine `startApp()` Methode, die das Midlet startet. Außerdem müssen die beiden Methoden `pauseApp()` und `destroyApp()` implementiert werden, die ebenfalls zur

Steuerung des Lebenszyklusses bzw. des Zustands des Midlets zuständig sind. Meist werden bei Midlets GUIs implementiert, die dazu notwendige Bibliothek ist `javax.microedition.lcdui`.

Ein kleines „Hello World“-Midlet, welches auf einem `Display` eine `TextBox` mit dem `"Hello World"`-Text anzeigt, sieht folgendermaßen aus:

```
import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;

public class HelloWorld extends MIDlet {
    private Display d;
    TextBox t = null;

    public HelloWorld () {
        d = Display.getDisplay(this);
        t = new TextBox("Test Title", "Hello World", 20, 0);
    }

    public void startApp() {
        d.setCurrent(t);
    }

    public void pauseApp() {
    }

    public void destroyApp(boolean b) {
    }
}
```

Abbildung 2: Das Beispielmidlet im Motorola i85s Simulator

Wenn die o.g. Voraussetzungen erfüllt sind, sind die folgenden Schritte notwendig, um aus dem Beispielmidlet eine auf einem Mobiltelefon laufende Anwendung zu machen (vgl. [15]):

1. *Schreiben des Programmtextes* in einem beliebigen Editor bzw. einer Entwicklungsumgebung.
2. *Kompilierung* mit dem Standard Java-Kompiler `javac` der J2ME.
3. *Pre-Verifizierung* mit dem Programm `preverify`.
4. *Erstellung eines JAR-Archivs* aus der pre-verifizierten Klasse.
5. *Erstellen eines Java Application Descriptors (JAD-Datei)* mit einem beliebigem Editor. Die JAD-Datei enthält verschiedene Metainformationen über das Midlet und muss zusammen mit dem JAR-Archiv vorliegen, um das Midlet auszuführen.
6. *Testen des Midlets* in einem Simulator, z.B. aus dem J2ME Wireless Toolkit ([22]).
7. *Übertragen von JAR- und JAD-Datei auf das Endgerät*, Tools dazu liefern die Gerätehersteller meist mit, bei Nokia Mobiltelefonen übernimmt das z.B. der Nokia Application Installer.
8. *Starten des Midlets* auf dem Endgerät.

Die Schritte 2 bis 6 können auch komfortabel mit der KToolbar ausgeführt werden. Mit diesem Tool ist auch das Anlegen der JAD-Datei sehr einfach möglich.

3.2 JControl

JControl ist eine extrem kompakte Java-Lösung für kleine eingebettete Systeme. Es wurde ursprünglich für 8-Bit Systeme mit sehr wenig Speicher entwickelt und besteht aus einer virtuellen Maschine, der JControlVM[5], Klassenbibliotheken und einer umfangreichen Entwicklungsumgebung, der JControl/DevelopmentSuite. JControl ist eine Entwicklung der Abteilung E.I.S. ([23]) der TU Braunschweig und der Domologic Home-Automation GmbH ([24]), welche aus der genannten Abteilung heraus entstanden ist.

Die Intention bei der Entwicklung von JControl war es, eine Java-Umgebung für eingebettete Systeme, speziell im Bereich der Home-Automation, zu schaffen, die mit weniger Ressourcen auskommt als die J2ME, speziell die CLDC. So wird JControl mit Systemanforderungen von ca. 50 kB ROM bzw. Flash, 2 kB RAM und 8-Bit CPU bei ca. 8 MHz von den Entwicklern auch im Bereich zwischen der CLDC der J2ME (mindestens 192 kB Speicher und 16-Bit CPU mit ca. 12 MHz) und Java Card (50 kB Speicher, 8-Bit CPU mit ca. 4 MHz) eingeordnet (vgl. [25]). Von der Komplexität her ist JControl also etwa mit Java Card vergleichbar, hat aber ein ganz anderes Anwendungsgebiet. Während Java Card für Chipkartenprozessoren gedacht ist, die z.B. Funktionen der Kryptographie übernehmen, ist mit JControl ein relativ breites Anwendungsspektrum möglich, welches von integrierten Steuerungsaufgaben, die der Benutzer gar nicht wahrnimmt (z.B. in einem herkömmlichen Kühlschrank), bis zu komplexen Benutzerschnittstellen mit Grafikdisplay reicht.

Das Zielsystem von JControl ist der ST7 Microcontroller von ST Microelectronics ([26]), der aus einem 68HC05 Kern von Motorola ([27]), der zur bekannten 68000er Reihe gehört, besteht. Er ist in vielen verschiedenen Konfigurationen erhältlich, die sich durch unterschiedliche Ausstattung an integriertem RAM und ROM und vor allem an den integrierten Schnittstellen unterscheiden. So ist dieser Prozessor für eine große Palette an eingebetteten Systemen geeignet.

An konkreten Produkten wurden bisher verschiedene universelle Steuerungseinheiten entwickelt, wie z.B. der JControl/Sticker ([28]), der aus einem ST7, einem 128x64 Pixel LCD und ein paar anderen elektronischen Bauteilen wie einem Tacktgeber besteht. Auf Basis solcher Einheiten wurden u.a. die Steuerungselektronik eines vernetzten Kühlschranks (vgl. [25]), ein Hardware Monitor für den I^2C-Bus von PC Mainboards (vgl. [29]) und ein Power-Line-Messsystem (vgl. [30]) realisiert.

[5] es wird zunächst nur die 8-Bit Variante beschrieben, die in manchen Veröffentlichungen auch JCVM8 genannt wird

Abbildung 3: JControl/Sticker (in Originalgröße, entnommen aus [28])

3.2.1 Die JControlVM und das JControl-API

Die virtuelle Maschine von JControl, die JControlVM, ist ein klassischer, aber sehr kleiner Java-Bytecode-Interpreter, welcher Ähnlichkeiten mit der KVM der J2ME aufweist, aber eben auch auf 8-Bit Systemen läuft. Die JControlVM wurde in Assembler geschrieben und ist daher sehr schnell. Zudem setzt sie nicht auf einem Betriebssystem auf, sondern beinhaltet selbst alle notwendigen Funktionen zur Hardwareansteuerung, was die Performance erneut steigert. Die JControlVM arbeitet mit Systemen ab 50 kB ROM bzw. Flash und 2 kB RAM (vgl. [25]) und bietet einen eigenen Garbage Collector. Durch einen angepassten Class-Loading-Mechanismus, der nicht nur das Laden, sondern auch das Entladen von Klassen ermöglicht, wird bei JControl im Gegensatz zu den Sun-Produkten die Implementierung mit einer minimalen Menge von 2 kB Arbeitsspeicher ermöglicht. Zudem übernimmt die JControlVM als Betriebssystemfunktion auch das Scheduling der Threads und bietet ein kleines Dateisystem für Java-Klassen und weitere Dateien wie z.B. Bilder.

Um die Größe der JControlVM und den Speicherbedarf zur Laufzeit weiter zu verringern, wurde auf Datentypen verzichtet, die größer als 16-Bit sind. Konkret bedeutet das, dass bei der Programmierung auf `int`-Datentypen mit 32-Bit, sowie auf die Datentypen `long`, `float` und `double` verzichtet werden muss. Das ist einerseits eine relativ große Einschränkung, andererseits bei der Entwicklung von kleinen eingebetteten Systemen durchaus zu verkraften.

Die Klassenbibliothek von JControl, das JControl-API, ist eine Untermenge der J2SE-API mit einigen Einschränkungen, aber auch einigen zusätzlichen Paketen wie solchen zur Ansteuerung der angeschlossenen Hardware. Neben Klassen zur Ansteuerung des Speichers, von Keyboards, dem Display, verschiedenen Kommunikationsschnittstellen (z.B. Infarot, RS232, CAN-Bus) stehen auch Systemklassen zur Verfügung, die das eigene Thread-Modell umsetzen, den Download-Modus zum Übertragen von Daten steuern und spezielle ma-

thematische Funktionen für die modifizierten Datentypen zur Verfügung stellen.

3.2.2 Die Grafikbibliothek Vole

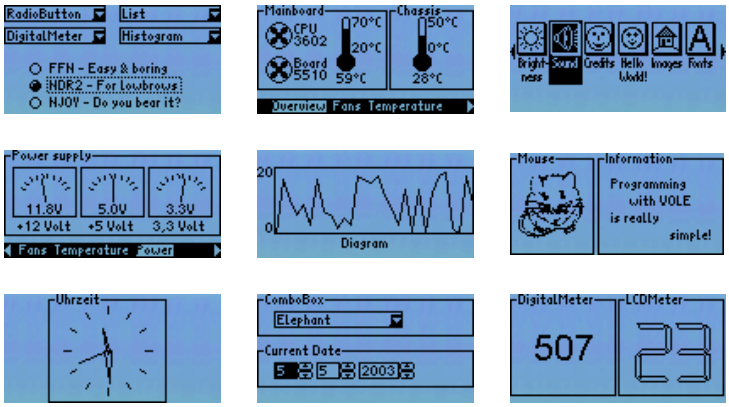

Abbildung 4: Verschiedene Screenshots von Vole-GUIs (entnommen aus [24])

Da JControl selbst nur sehr rudimentäre Methoden zur Grafikerzeugung auf angeschlossenen Displays zur Verfügung stellt, JControl-Geräte aber durchaus dazu gedacht sind, Benutzerschnittstellen zur Verfügung zu stellen, ist Vole entstanden (vgl. [29]). Vole ist ein sehr kompaktes GUI-Framework, welches neben den typischen Funktionen wie der Generierung von Buttons, Auswahlmenüs, Textfeldern und Checkboxen auch spezielle Funktionen für kleine eingebettete Steuerungs- und Überwachungsgeräte integriert. Vole kann so z.B. Graphen zeichnen und Messwerte in verschiedenen Formen visualisieren (s. Grafik mit Vole-Screnshots).

3.2.3 Die JControl-Entwicklungsumgebung JCManager

Der JCManager ist das zentrale Tool bei der JControl-Entwicklung. Es ist eine Java-Anwendung, die also auf vielen Plattformen läuft. Der JCManager vereint die Möglichkeiten der Simulation, des Erstellens von Transfer-Images und der Übertragung dieser Images auf JControl-Geräte mit zusätzlichen Tools zur Grafik-, Melodie- und Schriftartenbearbeitung.

Es ist vorgesehen, dass der Java-Code mit einem beliebigem Editor bzw. IDE geschrieben- und mit dem normalen `javac` Kompiler der J2SE kompiliert wird, ggf. auf Knopfdruck in der IDE. Mit dem JCManager werden dann

die weiteren Schritte vorgenommen. Es kann die Simulation eines JControl-Gerätes gestartet werden und vor allem kann aus den Klassendateien, sowie allen weiteren Daten wie Bildern, Melodien oder Schriftarten, eine Art Archiv gemacht werden, welches dann mit dem JCManager über die serielle Schnittstelle auf ein Gerät übertragen werden kann.

3.2.4 Praxisbeispiel: JControl-Entwicklung

Im folgenden soll analog zum J2ME-Beispiel ein „Hello World“-Programm für ein JControl-Gerät vorgestellt werden. Beim Quellcode (entnommen von [24]) dazu ist die einzige Besonderheit die Verwendung der importierten Klasse `jcontrol.io.Display` zur Displayansteuerung mit den auf ihr operierenden Methoden.

```
import jcontrol.io.Display;

public class HelloWorld {
  static Display lcd;

  public HelloWorld() {
    lcd = new Display();
    lcd.drawString("Hello World!", 42, 30);
    lcd.drawRect(20, 20, 88, 25);
    for (;;) {} // sleep well
  }

  public static void main(String[] args) {
    new HelloWorld();
  }
}
```

Hier noch einmal die Zusammenfassung des Entwicklungsablaufs:

1. *Schreiben des Programmtextes* in einem beliebigen Editor bzw. einer Entwicklungsumgebung.
2. *Kompilierung* mit dem Standard Java-Kompiler `javac` der J2ME.
3. *Anlegen eines Projekts im JCManager*
4. *Simulation der Anwendung* im JControl/Simulator
5. *Übertragen der Anwendung auf das JControl-Gerät* mit dem JCManager
6. *Starten der Anwendung* auf dem JControl-Gerät

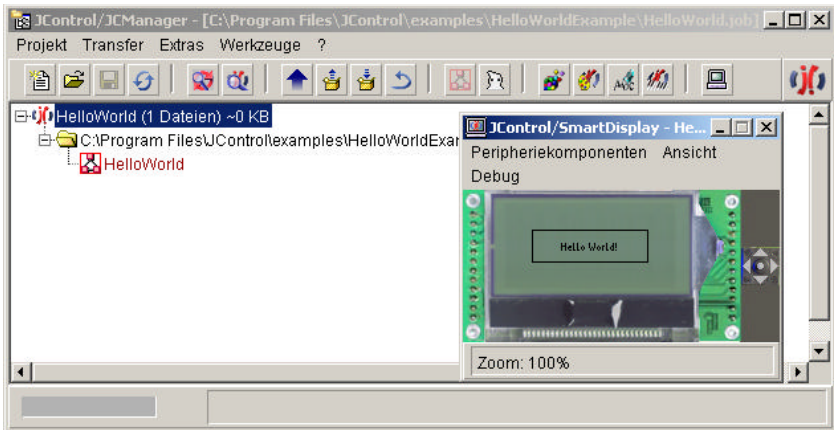

Abbildung 5: JCManager mit Simulation des „Hello World“ Beispiels

3.2.5 32-Bit JControl

Inzwischen wurde neben der 8-Bit Variante von JControl auch eine 32-Bit Version für größere Prozessoren entwickelt. Diese ist für größere eingebettete Geräte gedacht und läuft zur Zeit auf ColdFire MCF5206e Prozessoren von Motorola ([27]) mit einem Minimum von 2 MB DRAM (vgl. [3]). Es sind aber auch Portierungen für andere PLattformen, u.a. für Xilinx ML300 und PPC405 Prozessoren geplant. Bei der 32-Bit Variante übernimmt die virtuelle Maschine von JControl (JCVM32) nicht mehr die Betriebssystemfunktionen, sondern setzt auf einem Betriebssystem, z.B. uClinux, Linux oder Windows auf. Es handelt sich hierbei also um eine Hybridlösung aus einem in C geschriebenem Betriebssystem und darauf laufenden Java-Programmen. 32-Bit JControl ist somit deutlich flexiebler und skalierbarer als die 8-Bit Version. Für die 32-Bit Variante, die auch für größere Displays gedacht ist, kann sowohl Vole als GUI Bibliothek verwendet werden wie auch der JCManager zur Entwicklung.

Von Domologic wurden auf Basis des Motorola ColdFire Prozessors und JControl 32-Bit verschiedene Produkte zur Videobearbeitung, zur Touchscreen-Ansteuerung und für Netzwerkanwendungen realisiert. Z.B. existiert ein OEM IP Gateway ([31]) als Basis für Netzwerkgeräte mit HTML-Seitengenerator und es wurde eine Touch-Screen-Bedieneinheit für Außenkameras entwickelt, welche über ein großes LCD verfügt. Über diese Bedieneinheit können Kameras gesteuert, Videos angezeigt und einige Einstellungen vorgenommen werden. Zudem wird ein neuartiges, XML-basiertes GUI-System verwendet, welches aus einer XML-Datei Java-Code für ein GUI generiert.

4 Technologieübersicht

In diesem Kapitel soll noch einmal ein Überblick über die bisher betrachteten Technologien gegeben werden, sowie weitere Technologien in kürze vorgestellt werden.

Bisher wurde die J2ME mit den virtuellen Maschinen CDC HotSpot Implementation, KVM und CLDC HotSpot Implementation-, sowie JControl mit der JControlVM vorgestellt. Alle genannten virtuellen Maschinen sind allein nicht für Anwendungen geeignet, die harte Echtzeit voraussetzen. Die JControlVM in der 8-Bit Variante läuft ohne Betriebssystem und stellt diese Funktionen selbst zur Verfügung, die 16-Bit Version der JControlVM und die HotSpot-Implementierungen laufen auf Betriebssystemen und bei der KVM sind prinzipiell beide Betriebsarten möglich.

4.1 Interpreter und verschiedene Kompiler

Bei KVM und JControlVM handelt es sich um reine Bytecode-Interpreter. Die HotSpot-Implementierungen sind eine spezielle Form der so genannten just-in-time (JIT) Kompiler, die auf dem Endgerät Bytecode zu nativem Code kompilieren, bevor er ausgeführt wird. JIT-Kompiler haben u.a. den Nachteil, dass sie 4 bis 8 mal so viel Speicher benötigen wie reine Interpreter (vgl. [20]). Da so viel Speicher bei kleinen Geräten nicht zur Verfügung steht und der Kompiliervorgang bei kleinen Prozessoren zu merklichen Startverzögerungen führen würde, kompilierter Code aber deutliche Performancevorteile bei der Ausführung bietet, wurden von Sun, aber auch von anderen Forschern, die so genannten hot spot Kompiler entwickelt. Diese sind eine Mischung aus Interpreter und JIT-Kompiler. Unter einem hot spot versteht man einen Programmbereich oder Methoden, die besonders häufig aufgerufen werden. Getreu dem Motto „make the common case fast“ der Rechnerarchitekten, werden in einem hot spot Kompiler lediglich die hot spots, also die am häufigsten ausgeführten Programmbereiche kompiliert, der Rest des Bytecodes weiterhin interpretiert. Folglich müssen Entscheidungen getroffen werden, welche Programmteile häufig ausgeführt werden, also welche kompiliert werden sollen. Dieser Vorgang der Entscheidungsfindung wird von so genannten Profilern durchgeführt.

Wie in der Grafik zu sehen ist, besteht die CLDC HotSpot Implementation von Sun tatsächlich aus Interpreter, Profiler, Kompiler und Speicher für Bytecode und kompilierten Code. Zunächst führt der Interpreter Bytecode aus und sendet Meldungen über die Programmausführung an den Profiler. Dieser entscheidet, welche Programmteile kompiliert werden sollen und teilt die

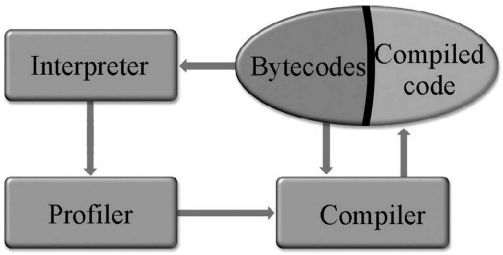

Abbildung 6: Die Architektur der CLDC HotSpot Implementation (entnommen aus [20])

Entscheidung dem Kompiler mit. Der Kompiler kompiliert Bytecode zu nativem Code, den er im Speicher ablegt. Steht nativer kompilierter Code zur Verfügung, wird dieser ausgeführt, ansonsten wird Bytecode interpretiert.

Ein französisches Forscherteam ([32]) schlägt eine Lösung vor, bei der der Vorgang der Kompilierung der hot spots auf einem speziellen externen Server stattfindet. So kann zwar das rechenintensive Kompilieren auf leistungsfähigen Servern durchgeführt werden und es wird kein Kompiler auf dem eingebetteten Endgerät benötigt, die Vorgehensweise impliziert aber, dass die Geräte über eine permanente und hinreichend schnelle Netzwerkverbindung verfügen müssen, damit die Server benutzt werden können. Eine solche ist aber nur bei wenigen eingebetteten Systemen vorhanden.

Auch wenn hot spot Kompiler gute Performance versprechen, benötigen sie doch deutlich mehr Speicher und auch mehr Rechenleistung als reine Interpreter. Aus diesem Grund wird bei kleinen Geräten, wie denen der 8-Bit JControl, sicherlich auch in Zukunft auf Interpretertechnologie gesetzt werden. Zwar möchte Sun in Zukunft seine HotSpot-Technologie auch in der CLDC einsetzen, diese benötigt allerdings einen leistungsfähigen 32-Bit Prozessor und wird kleinere Systeme nicht mehr unterstützen. Da die CLDC und im speziellen das MIDP als größten Markt Mobiltelefone hat, deren Hardwareausstattung bei wenigen Geräten (wie dem Nokia 6600 mit 6 MB Flash Speicher) schon jetzt größer ist, als die, die bei der Entwicklung der KVM eingeplant wurde, kann die CLDC HotSpot Implementation ohne weiteres auf den neuen Mobiltelefon-Generationen bzw. Smartphones eingesetzt werden und wird dort für mehr Performance sorgen, als es die KVM bietet.

Gute Performance und geringen Speicherbedarf versprechen reine Kompiler-Lösungen, bei denen nur kompilierter Code auf das Endgerät übertragen wird. Dieses Verfahren wird auch ahead-of-time (AOT) Kompilieren genannt. Hier wird allerdings bewusst auf die Plattformunabhängigkeit verzichtet, da nur nativer Code auf die Geräte kommt - eine virtuelle Maschine wird also nicht benötigt. Einen solchen Weg geht u.a. das GNU Compiler for the Java Pro-

gramming Language (GCJ) Projekt ([33]), welches Kompiler für verschiedene Plattformen entwickelt. Auf diesem Projekt setzt das Projekt jRate auf, welches an der University of California, Irvine (UCI) entwickelt wird (vgl. [34]). jRate hat das Ziel, mit Hilfe des GCJ die anfangs erwähnte Real-Time Specification for Java (RTSJ, [6]) umzusetzen und so Java bedingt echtzeitfähig zu machen.

Eine Mischung aus JIT- und AOT-Kompiler ist Jbed von der Esmertec AG ([35]). Jbed besteht aus dem Fast ByteCode Compiler (FastBCC), welches ein JIT-Kompiler ist und dem Static Host Compiler, der als AOT-Kompiler auf einem anderen Rechner als dem Zielsystem nativen Code kompiliert. Als Einsatzbereich von Jbed wird die MIDP angegeben. Esmertec hat noch eine andere, aufwendigere Technologie für virtuelle Maschinen, die sie Dynamic Adaptive Compiler (DAC) nennen und der hot spot JIT-Kompiler-Technik sehr ähnlich ist. Die virtuellen Maschinen mit DAC-Technik sind Jeode für das CDC (speziell für größere PDAs) und JeodeK für die CLDC.

4.2 Java-Prozessoren

Neben Interpreter- und Kompilerlösungen gibt es noch einen anderen Ansatz, Java in hoher Geschwindigkeit auszuführen, der zudem auch Echtzeitfähigkeiten verspricht. Es handelt sich um Microprozessoren, die direkt Java-Bytecode ausführen, also um Umsetzungen auf Hardwarebasis, die auch kein Betriebssystem benötigen. Der erste Java-Chip ist der aJ-100 Prozessor (vgl. [36]) von aJile Systems ([37]). Dieser 32-Bit Prozessor ist eine vollständige Java-Umsetzung in Silizium, die dazu verwendet werden kann, Echtzeitsysteme (nach RTSJ) aufzubauen oder MIDP-Anwendungen auszuführen.

Der aJ-100 verspricht, kein nichtdeterministisches Verhalten aufzuweisen, bietet Hardwareunterstützung für Java-Threads in Echtzeit sowie einen Garbage Collector. Für Echtzeitanwendungen steht ein gesonderter Speicherbereich zur Verfügung, in dem keine Garbage Collection stattfindet. In [38] beschreibt ein aJile Mitarbeiter, dass die Java-Prozessoren seiner Firma angeblich genauso schnell sind wie vergleichbare Prozessoren mit herkömmlicher, C-basierter Software, 5 mal schneller sind als Java-Interpreter auf solchen Prozessoren und zudem nur 1/20 der Energie benötigen.

Fertige universelle Geräte mit aJile-Prozessoren liefert Systronics Inc. ([39]). Diese haben einen ähnlichen Einsatzbereich wie die JControl-Geräte von Domologic, z.B. sind JStamp ([40]) und JControl/Stamp sehr ähnlich, nur dass die JControl/Stamp mit einem herkömmlichen Microprozessor- und die JStamp mit einem aJile Java-Prozessor arbeitet.

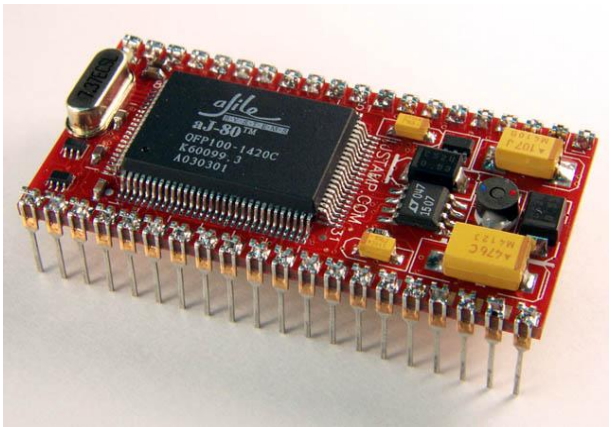

Abbildung 7: JStamp mit aJ-80 Java-Prozessor (entnommen aus [39])

Neben Java-Prozessoren bietet aJile seinen Java-Kern zur Lizensierung auch Herstellern an, die diesen in Systems-on-Chips als Java-Coprozessor integrieren wollen.

Java-Prozessoren scheinen viele Vorteile zu bringen, allerdings gibt es nur wenig Auswahl, so dass man nicht für jede Anwendung einen passenden Prozessor finden wird. Zudem kann man bei solchen Prozessoren nicht auf herkömmliche Programmiersprachen zurückgreifen. Aus diesen Gründen werden Java-Prozessoren wohl eher eine Nieschenanwendung bleiben und nur bei wenigen speziellen Geräten eingesetzt werden.

5 Ausblick

Ein detaillierter Vergleich aller Java-Technologien für eingebettete Systeme im Hinblick auf Echtzeitfähigkeit, Ressourcenverbrauch und vor allem Geschwindigkeit ist im Rahmen dieser Ausarbeitung leider nicht möglich. In den entsprechenden Papieren und Webseiten werden meist die eigenen Produkte jeweils so positiv dargestellt, dass ein Vergleich allein mit den angegebenen Quellen nicht möglich ist. Dazu wären viel umfassendere Recherchen- oder besser eigene Tests und Messungen notwendig.

Angesichts der jetzt schon großen Anzahl von verkauften J2ME-Geräten zeichnet es sich ab, dass die J2ME in Kombination mit herkömmlichen Prozessoren, gerade im Bereich von Mobiltelefonen, PDAs und ähnlichen Geräten, auch in Zukunft dominieren wird. Fraglich ist nur, in weit sich Java auch als Plattform für Applikationen wie z.B. Organizern auf diesen Geräten durchsetzen wird. Zudem werden durch immer größere Hardwareressourcen auch immer effizientere Java-Implementierungen möglich werden.

Für den Bereich kleiner eingebetteter Systeme, also für alles, was mit 8-Bit oder 16-Bit Prozessoren zu klein für die zukünftige Micro Edition ist, gibt es Raum für Java-Technologien wie JControl. Solche Implementierungen laufen auf kostengünstigen, kleinen Prozessoren, die es für eine Vielzahl von Anwendungsszenarien gibt. Oft kommt es dabei, wie z.B. in der Home-Automation, auch nicht auf harte Echtzeit an. Die in der Einleitung erwähnten Vorteile von Java können hier voll ausgenutzt werden und es ist zu erwarten, dass kleine Java-Implementierungen in zunehmendem Maße eingesetzt werden, entweder eigenständig oder in Kombination mit einem (Echtzeit-)Betiebssystem.

Literatur

[1] Broy, M.; Pree, W. Ein Wegweiser für Forschung und Lehre im Software-Engineering eingebetteter Systeme. *Informatik Spektrum*, Februar 2003.

[2] Flanagan, D. *Java Power Reference*. O'Reilly, 1999.

[3] Domologic Home-Automation GmbH. Kostenoptimierte Lösungen für frei konfigurierbare Touch-Panels und IP-Gateways mit JControl. 2003. http://www.domologic.de/download/pdf/konfigurationskonzepte_de.pdf.

[4] Java Community Process Program Homepage. http://www.jcp.org/.

[5] The Real-Time for Java Expert Group Homepage. http://www.rtj.org/.

[6] Bollella, G. et al. *The Real-Time Specification for Java*. Addison-Wesley, 2000. http://www.rtj.org/rtsj-V1.0.pdf.

[7] Hardin, D. S. Embedded and Real-Time Java. *IEEE Instrumentation & Measurement Magazine*, Juni 2000.

[8] Sun Microsystems, Inc. Datasheet Java 2 Platform, Micro Edition. 2002. http://java.sun.com/j2me/docs/j2me-ds.pdf.

[9] Java 2 Enterprise Edition Homepage. http://java.sun.com/j2ee/.

[10] Java 2 Standard Edition Homepage. http://java.sun.com/j2se/.

[11] Java 2 Micro Edition Homepage. http://java.sun.com/j2me/.

[12] Java Card Homepage. http://java.sun.com/products/javacard/.

[13] Connected Device Configuration Homepage. http://java.sun.com/products/cdc/.

[14] Sun Microsystems, Inc. Datasheet Java 2 Platform, Micro Edition (J2ME) Connected Device Configuration (CDC). 2003. .

[15] Eschweiler, S. *Das Einsteigerseminar - Java 2 Micro Edition*. Verlag Moderne Industrie, 2003.

[16] Sun Microsystems, Inc. White Paper CDC: An Application Framework for Personal Mobile Devices. Juni 2003. http://java.sun.com/products/cdc/wp/cdc-whitepaper.pdf.

[17] Connected Limited Device Configuration Homepage.
http://java.sun.com/products/cldc/.

[18] Sun Microsystems, Inc. J2ME Building Blocks for Mobile Devices - White Paper on KVM and the Connected, Limited Device Configuration (CLDC). Mai 2000.
http://java.sun.com/products/cldc/wp/KVMwp.pdf.

[19] Sun Microsystems, Inc. Connected Limited Device Configuration (CLDC) Specification Version 1.1. März 2003.
http://jcp.org/aboutJava/communityprocess/final/jsr139/.

[20] Sun Microsystems, Inc. White Paper The CLDC HotSpot Implementation Virtual Machine. 2002.
http://java.sun.com/products/cldc/wp/CLDC_HI_WhitePaper.pdf.

[21] Mobile Information Device Profile Homepage.
http://java.sun.com/products/midp/.

[22] J2ME Wireless Toolkit Homepage.
http://java.sun.com/products/j2mewtoolkit/.

[23] Abteilung E.I.S. TU Braunschweig Homepage.
http://www.cs.tu-bs.de/eis/.

[24] Domologic Home-Automation GmbH Homepage.
http://www.domologic.de/.

[25] Böhme, H.; Telkamp, G. JControl - Einfache Implementierung und Evaluierung von eingebetteten Anwendungen mit Java (Tagungsbeitrag 10. E.I.S.-Workshop). *Abteilung E.I.S., TU Braunschweig; Domologic Home-Automation GmbH*, April 2001.
http://www.cs.tu-bs.de/eis/people/boehme/2001%20JControl_EIS.pdf.

[26] ST Microelectronics Homepage. http://www.st.com/.

[27] Motorola Semiconductors Homepage. http://e-www.motorola.com/.

[28] Domologic Home-Automation GmbH. Datasheet JControl/Sticker. Oktober 2003.
http://www.jcontrol.org/download/datasheets/jcontrol_sticker.pdf.

[29] Klingauf, W.; Telkamp, G.; Böhme, H. Java-basierte Benutzeroberflächen für extrem kompakte eingebettete Systeme. *Abteilung E.I.S., TU Braunschweig; Domologic Home-Automation GmbH*, Oktober 2003.
http://www.cs.tu-bs.de/eis/people/klingauf/2003%20Vole_Klingauf.pdf.

[30] Böhme, H.; Klingauf, W.; Telkamp, G. JControl - Rapid Prototyping und Design Reuse mit Java (Tagungsbeitrag 11. E.I.S.-Workshop). *Abteilung E.I.S., TU Braunschweig; Domologic Home-Automation GmbH*, März 2003. http://www.cs.tu-bs.de/eis/people/klingauf/2003%20JControl_EIS.pdf.

[31] Domologic Home-Automation GmbH. Datasheet OEM IP Gateway. http://www.domologic.de/download/pdf/oem_ip_gateway_en.pdf.

[32] Delsart, B.; Joloboff, V.; Paire, E. JCOD: A Lightweight Modular Compilation Technology for Embedded Java. *Lecture Notes in Computer Science*, 2491, 2002.

[33] GNU Compiler for the Java Programming Language Homepage. http://gcc.gnu.org/java/.

[34] Corsaro, A.; Schmidt, D.C. The Design and Performance of the jRate Real-Time Java Implementation. *Lecture Notes in Computer Science*, 2519, 2002.

[35] Esmertec AG Homepage. http://www.esmertec.com/.

[36] aJile Systems. Datasheet aJ-100 Real-time Low Power Java Processor. November 2000. http://www.ajile.com/downloads/aJ100Datasheet_1.3.pdf.

[37] aJile Systems Homepage. http://www.ajile.com/.

[38] Hardin, D. S. Crafting a Java Virtual Machine in Silicon. *IEEE Instrumentation & Measurement Magazine*, März 2001.

[39] Systronix Inc. Homepage. http://www.systronix.com/.

[40] Systronix Inc. Datasheet JStamp Real-Time Native Java Module. März 2002. http://jstamp.systronix.com/Resource/jstamp_datasheet.pdf.

Abbildungsverzeichnis